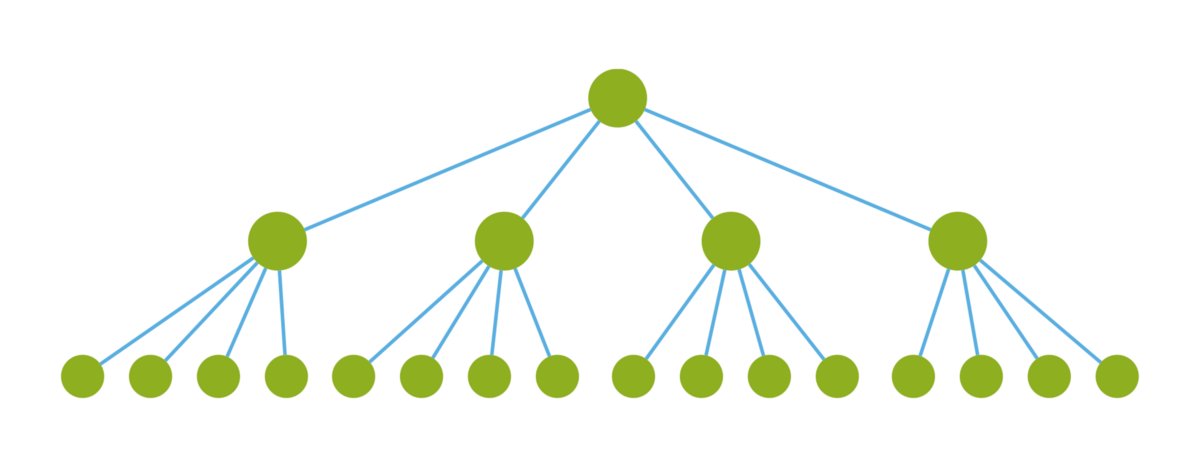
**Performance optimisations for React applications**

### What is the default render behaviour in React?

Let’s take a look at how React renders components

#### Initial render

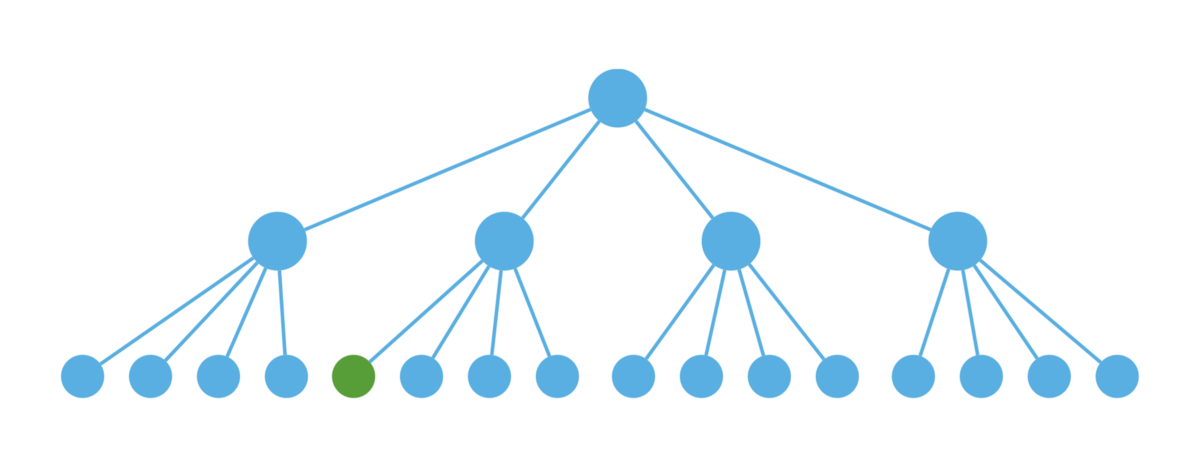
On the initial render we need the entire application to render  
(green = nodes that rendered)



Every node has rendered — this is good! Our application now represents our initial state

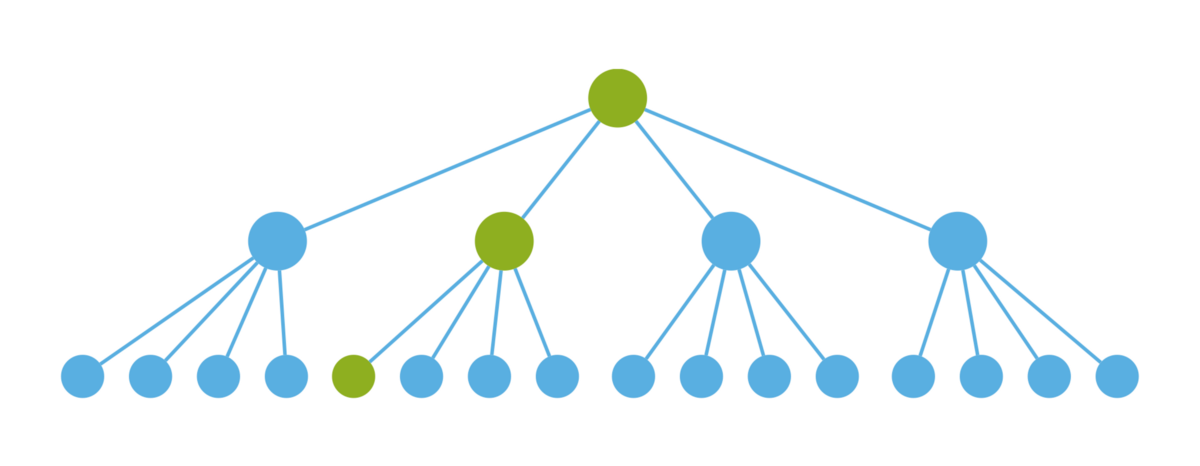
#### Proposed change

We want to update a piece of data. This change is only relevant to one leaf node



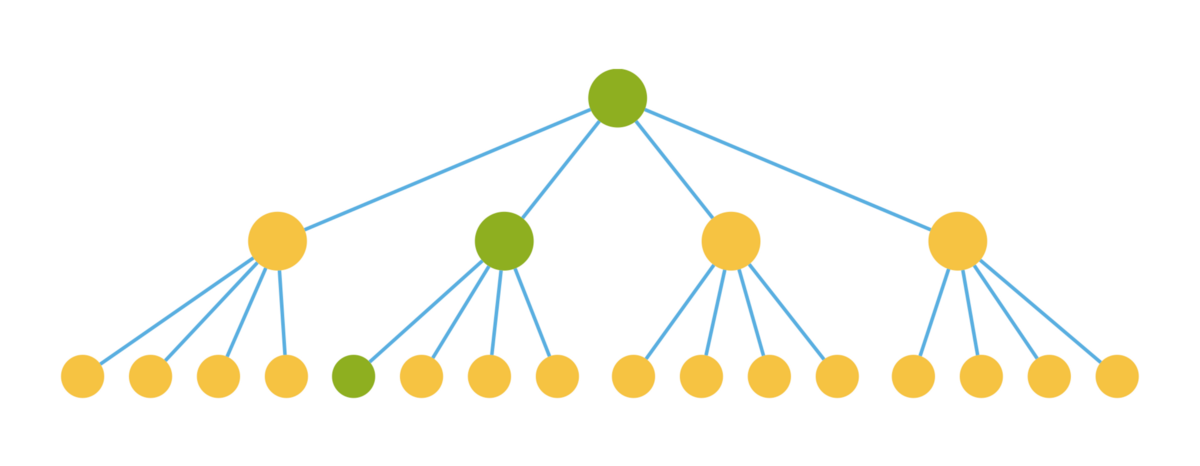
#### Ideal update

We want to only render the nodes that are along the critical path to our leaf node



#### Default behaviour

This is what React does if you do not tell it otherwise  
(orange = waste)



Oh no! All of our nodes have rendered.

# Benchmark React Components

When clicking the button, only the second array of ListItems should re-render — only they are affected by the change. Yet we’re experiencing slowness, so we want to investigate whether everything is as it should be.

We’ll start by installing React’s performance tools.

npm install --save-dev react-addons-perf

And then we can import it in our App.js:

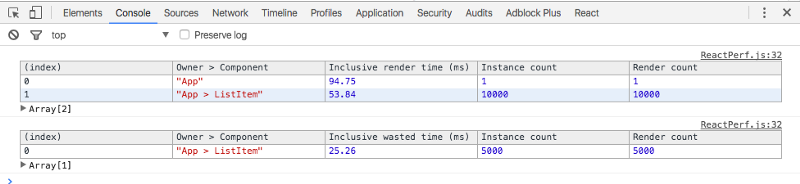
import Perf from 'react-addons-perf'

There are four Perf functions that we care about:

* Perf.start(): start measuring performance.
* Perf.stop()
* Perf.printExclusive(): prints total rendering time for components.
* Perf.printWasted(): prints wasted renders- we’ll get to this shortly.

We want to start measuring render time before our component starts updating- before we call *setState()*. Then we can stop the measurement and print the results using the lifecycle method *componentDidUpdate()*.

Here’s what our console will look like after clicking the button:



Our App renders took 94.75ms to render, and rendered only once.

Our ListItem component took 53.84ms, and rendered 10,000 times (this is including all instances of the components, not individual render time).

In the second table, we can see ‘wasted’ time — when the component re-rendered but nothing had actually changed.

We have 5,000 wasted ListItem renders. Not good.

These renders are ‘wasted’ in the sense that only the ListItems affected by the multiplier will have their render actually changed when we click the button. There’s no point in having them re-render.

### Fixing Wasted Renders

Fortunately, React furnishes us with a handy lifecycle method called shouldComponentUpdate() for these types of situations.

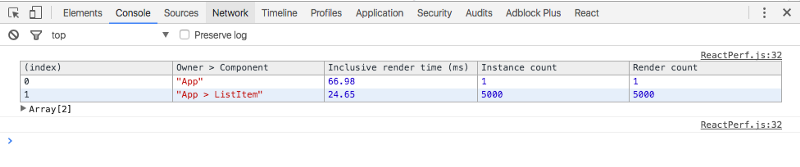
It gives us fine-grained control of when our component will re-render.

We can check for certain conditions, returning a boolean that React uses to determine whether it will call render() on the component, or leave it as is.

If the text prop hasn’t changed, there’s no reason to change the component’s appearance via a re-render.

Note that we don’t use the nextState argument in shouldComponentUpdate()- I left it there for reference.

Now let’s re-run our Perf.



We’ve cut our total render time down by 30ms, and eliminated all wasteful renders.

### Using Perf in the Real World

Most of the components you’ll be benchmarking won’t be nearly this simple. But the Perf tools are a valuable way to identify which components are problems.

### Benchmarking Initial Renders

Sometimes you want to not just benchmark when the component updates, but also how long it takes to render completely the first time around.

In short, we want to measure the time between componentWillMount()and componentDidMount().

Perf.start() can’t be placed in the componentWillMount(),however, so we’ll have to be a little more manual.

|  |
| --- |
| componentWillMount() { |
|  | window.performance.mark('App') |
|  | } |
|  |  |
|  | componentDidMount() { |
|  | console.log(window.performance.now('App')) |
|  | } |

This will give you a logged out time in milliseconds- not as fancy as Perf, but a useful metric for how long it takes to boot up your app.